9 Creating a Web API for mobile and client applications using MVC

This chapter covers

- Creating a Web API controller to return JSON to clients
- Using attribute routing to customize your URLs
- Generating a response using content negotiation
- Applying common conventions with the `[ApiController]` attribute

In the previous five chapters, you've worked through each layer of a server-side rendered ASP.NET Core application, using Razor Pages to render HTML to the browser. In this chapter you'll see a different take on an ASP.NET Core application. Instead of using Razor Pages, we'll explore Web APIs, which serve as the backend for client-side SPAs and mobile apps.

You can apply much of what you've already learned to Web APIs; they use the same MVC design pattern, and the concepts of routing, model binding, and validation all carry through. The differentiation from traditional web applications is primarily in the *view* part of MVC. Instead of returning HTML, they return data as JSON or XML, which client applications use to control their behavior or update the UI.

In this chapter you'll learn how to define controllers and actions and see how similar they are to the Razor Pages and controllers you already know. You'll learn how to create an API model to return data and HTTP status codes in response to a request, in a way that client apps can understand.

After exploring how the MVC design pattern applies to Web APIs, you'll see how a related topic works with Web APIs: routing. We'll look at how *attribute routing* reuses many of the same concepts from chapter 5 but applies them to your action methods rather than to Razor Pages.

One of the big features added in ASP.NET Core 2.1 was the `[ApiController]` attribute. This attribute applies several common conventions used in Web APIs, which reduces the amount of code you must write yourself. In section 9.5 you'll learn how automatic `400 Bad Requests` for invalid requests, model-binding parameter inference, and `ProblemDetails` objects can make building APIs easier and more consistent.

You'll also learn how to format the API models returned by your action methods using content negotiation, to ensure you generate a response that the calling client can understand. As part of this, you'll learn how to add support for additional format types, such as XML, so that you can generate XML responses if the client requests it.

One of the great aspects of ASP.NET Core is the variety of applications you can create with it. The ability to easily build a generalized HTTP Web API presents the possibility of using ASP.NET Core in a greater range of situations than can be achieved with traditional web apps alone. But *should* you build a Web API and why? In the first section of this chapter, I'll go over some of the reasons why you might or might not want to create a Web API.

9.1 What is a Web API and when should you use one?

Traditional web applications handle requests by returning HTML to the user, which is displayed in a web browser. You can easily build applications of this nature using Razor Pages to generate HTML with Razor templates, as you've seen in recent chapters. This approach is common and well understood, but the modern application developer also has a number of other possibilities to consider, as shown in figure 9.1.

Client-side single-page applications (SPAs) have become popular in recent years with the development of frameworks such as Angular, React, and Vue. These frameworks typically use JavaScript that runs in a user's web browser to generate the HTML they see and interact with. The server sends this initial JavaScript to the browser when the user first reaches the app. The user's browser loads the JavaScript and initializes the SPA before loading any application data from the server.

> **NOTE** Blazor WebAssembly is an exciting new SPA framework. Blazor lets you write an SPA that runs in the browser just like other SPAs, but it uses C# and Razor templates instead of JavaScript by using the new web standard, WebAssembly. I don't cover Blazor in this book, so to find out more, I recommend *Blazor in Action*, by Chris Sainty (Manning, 2021).

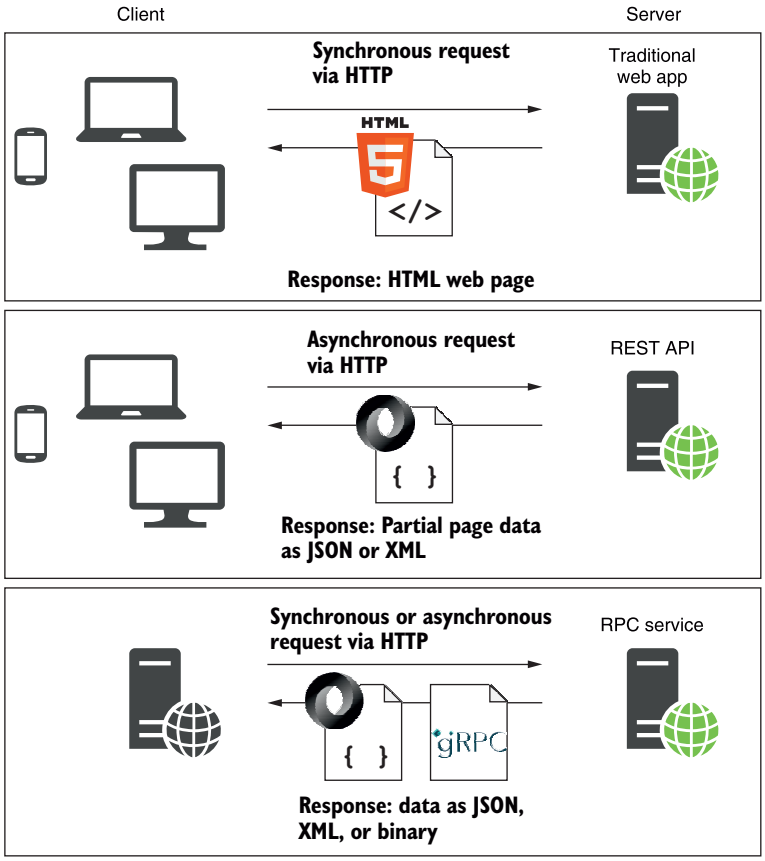

Figure 9.1 Modern developers have to consider a number of different consumers of their applications. As well as traditional users with web browsers, these could be SPAs, mobile applications, or other apps.

Once the SPA is loaded in the browser, communication with a server still occurs over HTTP, but instead of sending HTML directly to the browser in response to requests, the server-side application sends data (normally in a format such as JSON) to the client-side application. The SPA then parses the data and generates the appropriate HTML to show to a user, as shown in figure 9.2. The server-side application endpoint that the client communicates with is sometimes called a *Web API*.

> **DEFINITION** A *Web API* exposes multiple URLs that can be used to access or change data on a server. It's typically accessed using HTTP.

These days, mobile applications are common and are, from the server application's point of view, similar to client-side SPAs. A mobile application will typically communicate with a server application using an HTTP Web API, receiving data in a common

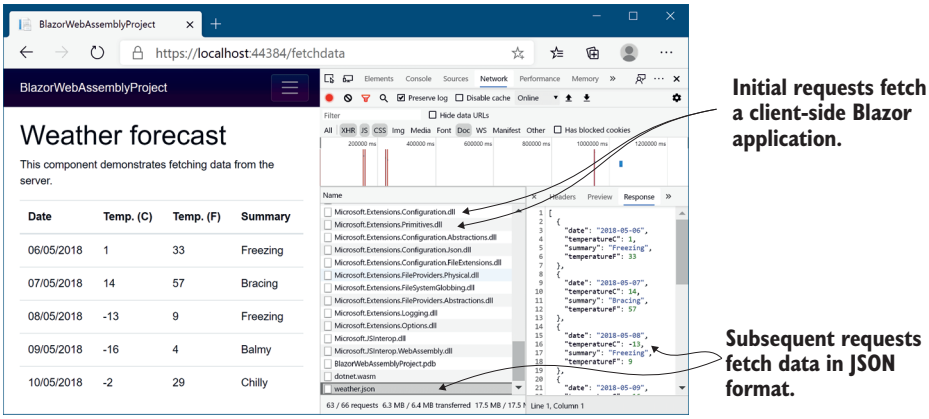

Figure 9.2 A sample client-side SPA using Blazor WebAssembly. The initial requests load the SPA files into the browser, and subsequent requests fetch data from a Web API, formatted as JSON.

format, such as JSON, just like an SPA. It then modifies the application's UI depending on the data it receives.

One final use case for a Web API is where your application is designed to be partially or solely consumed by other backend services. Imagine you've built a web application to send emails. By creating a Web API, you can allow other application developers to use your email service by sending you an email address and a message. Virtually all languages and platforms have access to an HTTP library they could use to access your service from code.

This is all there is to a Web API. It exposes a number of endpoints (URLs) that client applications can send requests to and retrieve data from. These are used to power the behavior of the client apps, as well as to provide all the data the client apps need to display the correct interface to a user.

Whether you need or want to create a Web API for your ASP.NET Core application depends on the type of application you want to build. If you're familiar with client-side frameworks, you need to develop a mobile application, or you already have an SPA build-pipeline configured, you'll most likely want to add Web APIs that they can use to access your application.

One of the selling points of using a Web API is that it can serve as a generalized backend for all your client applications. For example, you could start by building a client-side application that uses a Web API. Later you could add a mobile app that uses the same Web API, with little or no modification required to your ASP.NET Core code.

If you're new to web development, you have no need to call your application from outside a web browser, or you don't want or need the effort involved in configuring a client-side application, you probably won't need Web APIs initially. You can stick to generating your UI using Razor Pages and you'll no doubt be highly productive!

NOTE Although there has been an industry shift toward client-side frameworks, server-side rendering using Razor is still relevant. Which approach you choose will depend largely on your preference for building HTML applications in the traditional manner versus using JavaScript on the client.

Having said that, adding Web APIs to your application isn't something you have to worry about ahead of time. Adding them later is simple, so you can always ignore them initially and add them as the need arises. In many cases, this will be the best approach.

SPAs with ASP.NET Core

The cross-platform and lightweight design of ASP.NET Core means it lends itself well to acting as a backend for your SPA framework of choice. Given the focus of this book and the broad scope of SPAs in general, I won't be looking at Angular, React, or other SPAs here. Instead, I suggest checking out the resources appropriate to your chosen SPA. Books are available from Manning for all the common client-side JavaScript frameworks, as well as Blazor:

- *React in Action* by Mark Tielens Thomas (Manning, 2018)
- *Angular in Action* by Jeremy Wilken (Manning, 2018)
- *Vue.js in Action* by Erik Hanchett with Benjamin Listwon (Manning, 2018)
- *Blazor in Action*, by Chris Sainty (Manning, 2021)

Once you've established that you need a Web API for your application, creating one is easy, as it's built in to ASP.NET Core. In the next section you'll see how to create a Web API project and your first API controller.

9.2 Creating your first Web API project

In this section you'll learn how to create an ASP.NET Core Web API project and create your first Web API controllers. You'll see how to use controller action methods to handle HTTP requests, and how to use `ActionResults` to generate a response.

Some people think of the MVC design pattern as only applying to applications that directly render their UI, like the Razor views you've seen in previous chapters. In ASP.NET Core, the MVC pattern applies equally well when building a Web API, but the *view* part of MVC involves generating a *machine*-friendly response rather than a *user*-friendly response.

As a parallel to this, you create Web API controllers in ASP.NET Core in the very same way you create traditional MVC controllers. The only thing that differentiates them from a code perspective is the type of data they return—MVC controllers typically return a `ViewResult`; Web API controllers generally return raw .NET objects from their action methods, or an `IActionResult` such as `StatusCodeResult`, as you saw in chapter 4.

NOTE This is different from the previous version of ASP.NET, where the MVC and Web API stacks were completely independent. ASP.NET Core unifies the two stacks into a single approach (and adds Razor Pages to the mix), which makes using any option in a project painless!

To give you an initial taste of what you're working with, figure 9.3 shows the result of calling a Web API endpoint from your browser. Instead of a friendly HTML UI, you receive data that can be easily consumed in code. In this example, the Web API returns a list of `string` fruit names as JSON when you request the URL `/fruit`.

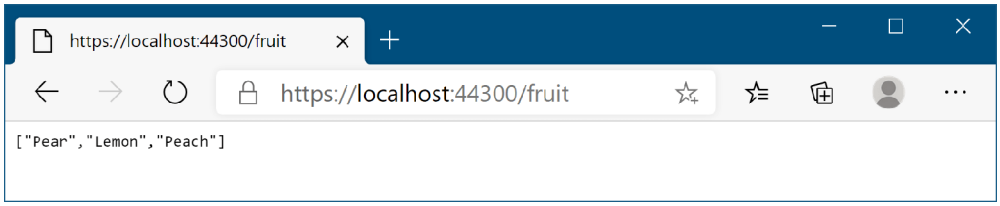

Figure 9.3 Testing a Web API by accessing the URL in the browser. A `GET` request is made to the `/fruit` URL, which returns a `List<string>` that has been JSON-encoded into an array of strings.

TIP Web APIs are normally accessed from code by SPAs or mobile apps, but by accessing the URL in your web browser directly, you can view the data the API is returning.

ASP.NET Core 5.0 apps also include a useful endpoint for testing and exploring your Web API project in development called Swagger UI, as shown in figure 9.4. This lets you browse the endpoints in your application, view the expected responses, and experiment by sending requests.

NOTE Swagger UI is based on the industry standard OpenAPI specification (previously called Swagger, www.openapis.org), which is enabled by default in Web API apps. OpenAPI provides a way of documenting your API, so that you can automatically generate clients for interacting with it in dozens of different languages. For more on OpenAPI and Swagger in ASP.NET Core apps, see Microsoft's documentation: http://mng.bz/QmjR.

You can create a new Web API project in Visual Studio using the same New Project process you saw in chapter 2. Create a new ASP.NET Core application providing a project name, and, when you reach the New Project dialog box, select the ASP.NET Core Web API template, as shown in figure 9.5. If you're using the CLI, you can create a similar template using `dotnet new webapi -o WebApplication1`.

The API template configures the ASP.NET Core project for Web API controllers only. This configuration occurs in the Startup.cs file, as shown in listing 9.1. If you compare this template to your Razor Pages projects, you'll see that the Web API project

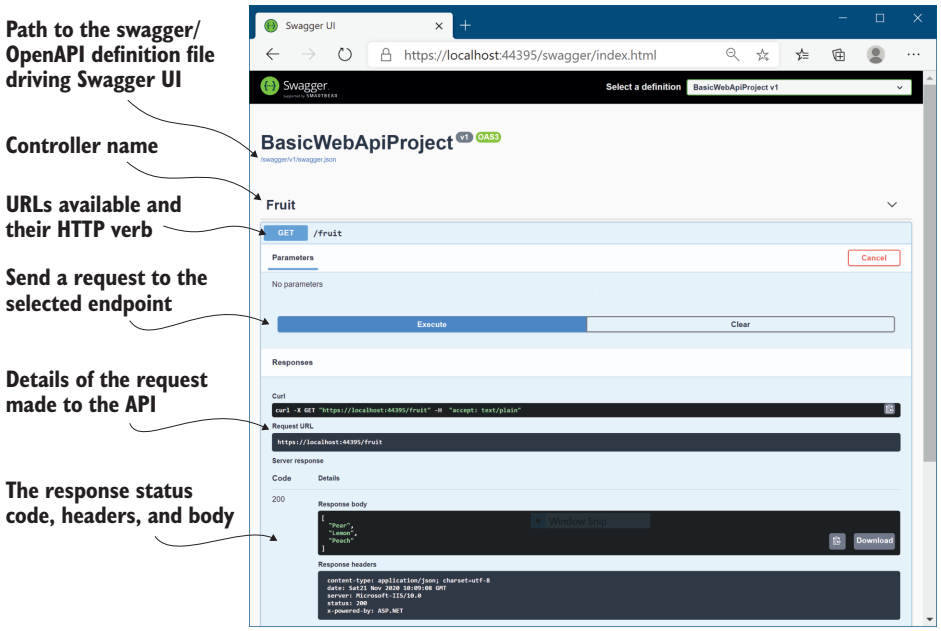

Figure 9.4 The default ASP.NET Core 5.0 Web API templates are configured to use Swagger UI. Swagger UI provides a convenient web page for exploring and interacting with your Web API. By default, this UI is only enabled in development, but you can also enable it in production if you wish.

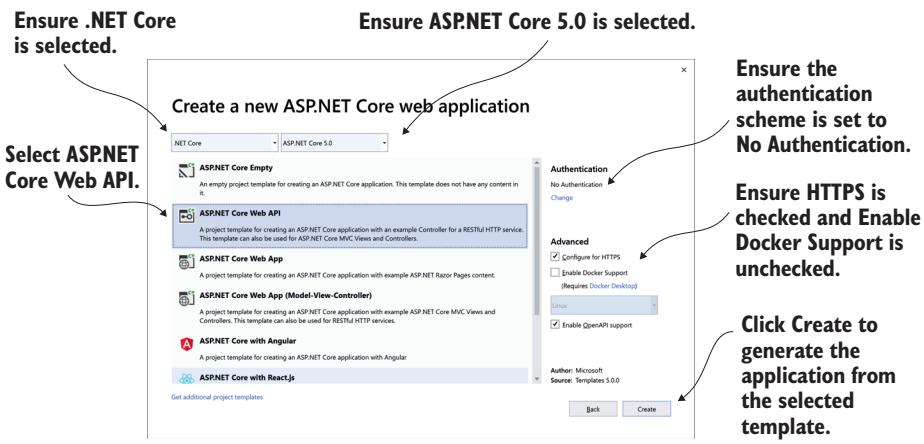

Figure 9.5 The web application template screen. This screen follows on from the Configure Your Project dialog box and lets you customize the template that will generate your application. Choose the ASP.NET Core Web API template to create a Web API project.

uses `AddControllers()` instead of `AddRazorPages()` in the `ConfigureServices` method. Also, the API controllers are added instead of Razor Pages by calling `MapControllers()` in the `UseEndpoints` method. The default Web API template also adds the Swagger services and endpoints required by the Swagger UI shown previously in figure 9.4. If you're using both Razor Pages and Web APIs in a project, you'll need to add all these method calls to your application.

Listing 9.1 The `Startup` class for the default Web API project

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddControllers();
        services.AddSwaggerGen(c =>
        {
            c.SwaggerDoc("v1", new OpenApiInfo {
                Title = "DefaultApiTemplate", Version = "v1" });
        });
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
            app.UseSwagger();
            app.UseSwaggerUI(c => c.SwaggerEndpoint(
              "/swagger/v1/swagger.json", "DefaultApiTemplate v1"));
        }

        app.UseHttpsRedirection();
        app.UseRouting();
        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapControllers();
        });
    }
}
```

Adds services required to generate the Swagger/OpenAPI specification document

AddControllers adds the necessary services for API controllers to your application.

Adds Swagger UI middleware for exploring your Web API

MapControllers configures the API controller actions in your app as endpoints.

The Startup.cs file in listing 9.1 instructs your application to find all API controllers in your application and to configure them in the `EndpointMiddleware`. Each action method becomes an endpoint and can receive requests when the `RoutingMiddleware` maps an incoming URL to the action method.

NOTE You don't have to use separate projects for Razor Pages and Web API controllers, but I prefer to do so where possible. There are certain aspects (such as error handling and authentication) that are made easier by taking this approach. Of course, running two separate applications has its own difficulties!

You can add a Web API controller to your project by creating a new .cs file anywhere in your project. Traditionally these are placed in a folder called Controllers, but that's not a technical requirement. Listing 9.2 shows the code that was used to create the Web API demonstrated in figure 9.3. This trivial example highlights the similarity to traditional MVC controllers.

Listing 9.2 A simple Web API controller

```
[ApiController]
public class FruitController : ControllerBase
{
    List<string> _fruit = new List<string>
    {
        "Pear",
        "Lemon",
        "Peach"
    };

    [HttpGet("fruit")]
    public IEnumerable<string> Index()
    {
        return _fruit;
    }
}
```

Web API controllers typically use the [ApiController] attribute to opt in to common conventions.

The ControllerBase class provides several useful functions for creating IActionResults.

The data returned would typically be fetched from the application model in a real app.

The [HttpGet] attribute defines the route template used to call the action.

The name of the action method, Index, isn't used for routing. It can be anything you like.

The controller exposes a single action method that returns the list of fruit.

Web APIs typically use the `[ApiController]` attribute (introduced in .NET Core 2.1) on API controllers and derive from the `ControllerBase` class. The base class provides several helper methods for generating results, and the `[ApiController]` attribute automatically applies some common conventions, as you'll see in section 9.5.

> **TIP** There is also a `Controller` base class, which is typically used when you use MVC controllers with Razor views. That's not necessary for Web API controllers, so `ControllerBase` is the better option.

In listing 9.2 you can see that the action method, `Index`, returns a list of strings directly from the action method. When you return data from an action like this, you're providing the API model for the request. The client will receive this data. It's formatted into an appropriate response, a JSON representation of the list in the case of figure 9.3, and sent back to the browser with a `200 OK` status code.

> **TIP** ASP.NET Core formats data as JSON by default. You'll see how to format the returned data in other ways in section 9.6.

The URL at which a Web API controller action is exposed is handled in the same way as for traditional MVC controllers and Razor Pages—using routing. The `[HttpGet ("fruit")]` attribute applied to the `Index` method indicates the method should use the route template `"fruit"` and should respond to HTTP `GET` requests. You'll learn more about attribute routing in section 9.5.

In listing 9.2, data is returned directly from the action method, but you don't *have* to do that. You're free to return an `ActionResult` instead, and often this is required. Depending on the desired behavior of your API, you may sometimes want to return data, and other times you may want to return a raw HTTP status code, indicating whether the request was successful. For example, if an API call is made requesting details of a product that does not exist, you might want to return a `404 Not Found` status code.

Listing 9.3 shows an example of just such a case. It shows another action on the same `FruitController` as before. This method exposes a way for clients to fetch a specific fruit by an `id`, which we'll assume for this example is its index in the list of `_fruit` you defined in the previous listing. Model binding is used to set the value of the `id` parameter from the request.

> **NOTE** API controllers use model binding, as you saw in chapter 6, to bind action method parameters to the incoming request. Model binding and validation work in exactly the same way as for Razor Pages: you can bind the request to simple primitives, as well as to complex C# objects. The only difference is that there isn't a `PageModel` with `[BindProperty]` properties—you can *only* bind to action method parameters.

Listing 9.3 A Web API action returning `IActionResult` to handle error conditions

```
[HttpGet("fruit/{id}")]
public ActionResult<string> View(int id)
{
    if (id >= 0 && id < _fruit.Count)
    {
        return _fruit[id];
    }
    return NotFound();
}
```

Defines the route template for the action method

The action method returns an ActionResult<string>, so it can return a string or an ActionResult.

An element can only be returned if the id value is a valid _fruit element index.

Returning the data directly will return the data with a 200 status code.

NotFound returns a NotFoundResult, which will send a 404 status code.

In the successful path for the action method, the `id` parameter has a value greater than 0 and less than the number of elements in `_fruit`. When that's true, the value of the element is returned to the caller. As in listing 9.2, this is achieved by simply returning the data directly, which generates a 200 status code and returns the element in the response body, as shown in figure 9.6. You could also have returned the data using an `OkResult`, by returning `Ok(_fruit[id])`, using the `Ok` helper method[1] on the `ControllerBase` class—under the hood, the result is identical.

[1] Some people get uneasy when they see the phrase "helper method," but there's nothing magic about the `ControllerBase` helpers—they're shorthand for creating a new `IActionResult` of a given type. You don't have to take my word for it, though. You can always view the source code for the base class on GitHub at http://mng.bz/goG8.

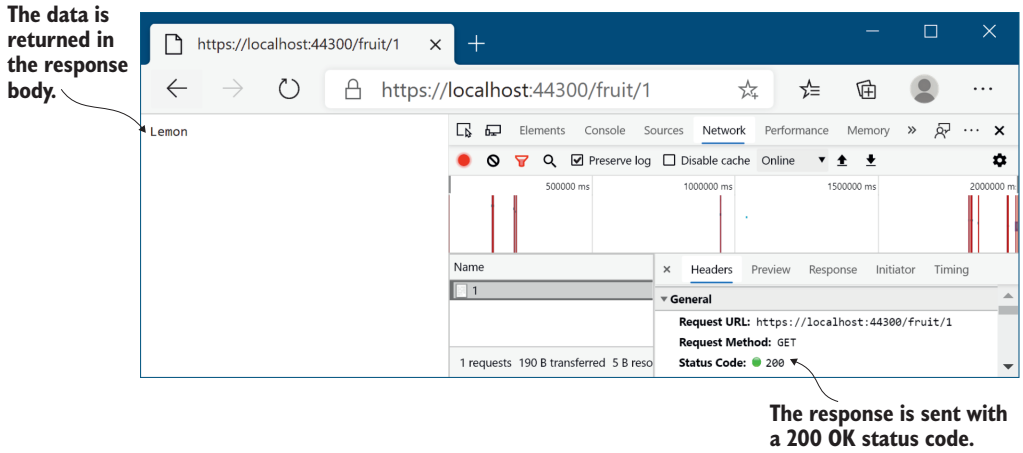

Figure 9.6 Data returned from an action method is serialized into the response body, and it generates a response with status code `200 OK`.

If the `id` is outside the bounds of the `_fruit` list, the method calls `NotFound` to create a `NotFoundResult`. When executed, this method generates a `404 Not Found` status code response. The `[ApiController]` attribute automatically converts the response into a standard `ProblemDetails` instance, as shown in figure 9.7.

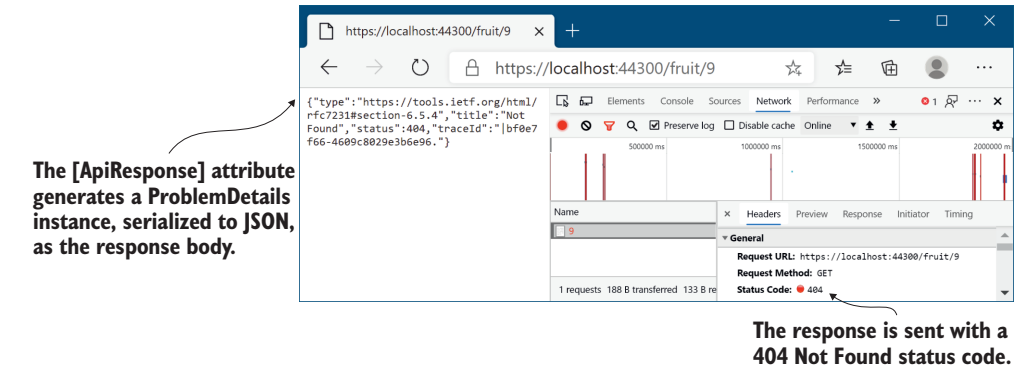

Figure 9.7 The `[ApiController]` attribute converts error responses (in this case a `404` response) into the standard `ProblemDetails` format.

> **DEFINITION** `ProblemDetails` is a web specification for providing machine-readable errors for HTTP APIs. You'll learn more about them in section 9.5.

One aspect you might find confusing from listing 9.3 is that for the successful case, we return a `string`, but the method signature of `View` says we return an `ActionResult<string>`. How is that possible? Why isn't it a compiler error?

The generic `ActionResult<T>` uses some fancy C# gymnastics with implicit conversions to make this possible.[2] Using `ActionResult<T>` has two benefits:

- You can return either an instance of `T` *or* an `ActionResult` implementation like `NotFoundResult` from the same method. This can be convenient, as in listing 9.3.
- It enables better integration with ASP.NET Core's OpenAPI support.

You're free to return any type of `ActionResult` from your Web API controllers, but you'll commonly return `StatusCodeResult` instances, which set the response to a specific status code, with or without associated data. `NotFoundResult` and `OkResult` both derive from `StatusCodeResult`, for example. Another commonly used status code is `400 Bad Request`, which is normally returned when the data provided in the request fails validation. This can be generated using a `BadRequestResult`. In many cases the `[ApiController]` attribute can automatically generate `400` responses for you, as you'll see in section 9.5.

> **TIP** You learned about various `ActionResults` in chapter 4. `BadRequestResult`, `OkResult`, and `NotFoundResult` all inherit from `StatusCodeResult` and set the appropriate status code for their type (200, 404, and 400, respectively). Using these wrapper classes makes the intention of your code clearer than relying on other developers to understand the significance of the various status code numbers.

Once you've returned an `ActionResult` (or other object) from your controller, it's serialized to an appropriate response. This works in several ways, depending on

- The formatters that your app supports
- The data you return from your method
- The data formats the requesting client can handle

You'll learn more about formatters and serializing data in section 9.6, but before we go any further, it's worth zooming out a little and exploring the parallels between traditional server-side rendered applications and Web API endpoints. The two are similar, so it's important to establish the patterns that they share and where they differ.

9.3 Applying the MVC design pattern to a Web API

In the previous version of ASP.NET, Microsoft commandeered the generic term "Web API" to create the ASP.NET Web API framework. This framework, as you might expect, was used to create HTTP endpoints that could return formatted JSON or XML in response to requests.

The ASP.NET Web API framework was completely separate from the MVC framework, even though it used similar objects and paradigms. The underlying web stacks for them were completely different beasts and couldn't interoperate.

---

[2] You can see how this is achieved in the source code: http://mng.bz/5j27.

In ASP.NET Core, that all changed. You now have a *single* framework that you can use to build both traditional web applications and Web APIs. The same underlying framework is used in conjunction with Web API controllers, Razor Pages, and MVC controllers with views. You've already seen this yourself; the Web API `FruitController` you created in section 9.2 looks very similar to the MVC controllers you've seen fleetingly in previous chapters.

Consequently, even if you're building an application that consists entirely of Web APIs, using no server-side rendering of HTML, the MVC design pattern still applies. Whether you're building traditional web applications or Web APIs, you can structure your application virtually identically.

By now you're, I hope, nicely familiar with how ASP.NET Core handles a request. But just in case you're not, figure 9.8 shows how the framework handles a typical

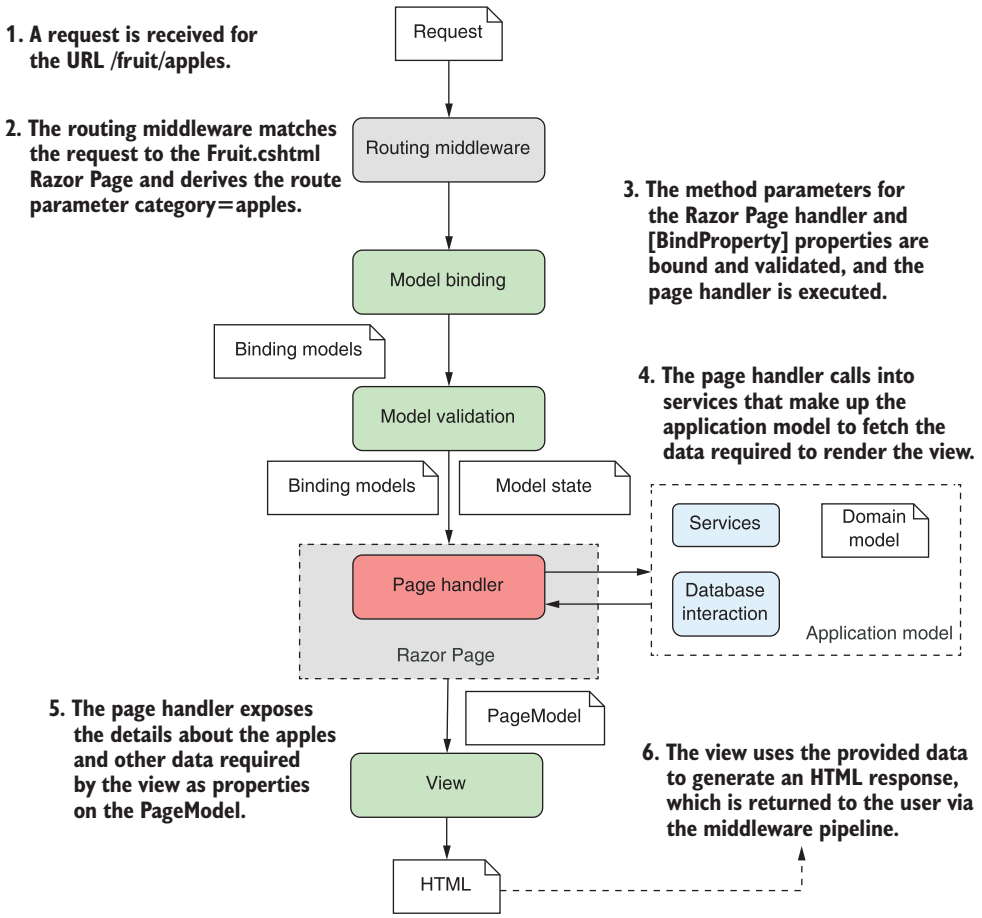

Figure 9.8 Handling a request to a traditional Razor Pages application, in which the view generates an HTML response that's sent back to the user. This diagram should be very familiar by now!

Razor Pages request after it passes through the middleware pipeline. This example shows how a request to view the available fruit on a traditional grocery store website might look.

The `RoutingMiddleware` routes the request to view all the fruit listed in the `apples` category to the Fruit.cshtml Razor Page. The `EndpointMiddleware` then constructs a binding model, validates it, sets it as a property on the Razor Page's `PageModel`, and sets the `ModelState` property on the `PageModel` base class with details of any validation errors. The page handler interacts with the application model by calling into services, talking to a database, and fetching any necessary data.

Finally, the Razor Page executes its Razor view using the `PageModel` to generate the HTML response. The response returns through the middleware pipeline and out to the user's browser.

How would this change if the request came from a client-side or mobile application? If you want to serve machine-readable JSON instead of HTML, what is different? As shown in figure 9.9, the answer is "very little." The main changes are related to switching from Razor Pages to controllers and actions, but as you saw in chapter 4, both approaches use the same general paradigms.

As before, the routing middleware selects an endpoint to invoke based on the incoming URL. For API controllers this is a controller and action instead of a Razor Page.

After routing comes model-binding, in which the binder creates a binding model and populates it with values from the request. Web APIs often *accept* data in more formats than Razor Pages, such as XML, but otherwise the model-binding process is the same as for the Razor Pages request. Validation also occurs in the same way, and the `ModelState` property on the `ControllerBase` base class is populated with any validation errors.

> **NOTE** Web APIs use *input formatters* to accept data sent to them in a variety of formats. Commonly these formats are JSON or XML, but you can create input formatters for any sort of type, such as CSV. I show how to enable the XML input formatter in section 9.6. You can see how to create a custom input formatter at http://mng.bz/e5gG.

The action method is the equivalent of the Razor Page handler; it interacts with the application model in exactly the same way. This is an important point; by separating the behavior of your app into an application model, instead of incorporating it into your pages and controllers themselves, you're able to reuse the business logic of your application with multiple UI paradigms.

> **TIP** Where possible, keep your page handlers and controllers as simple as practicable. Move all your business logic decisions into the services that make up your application model, and keep your Razor Pages and API controllers focused on the mechanics of interacting with a user.

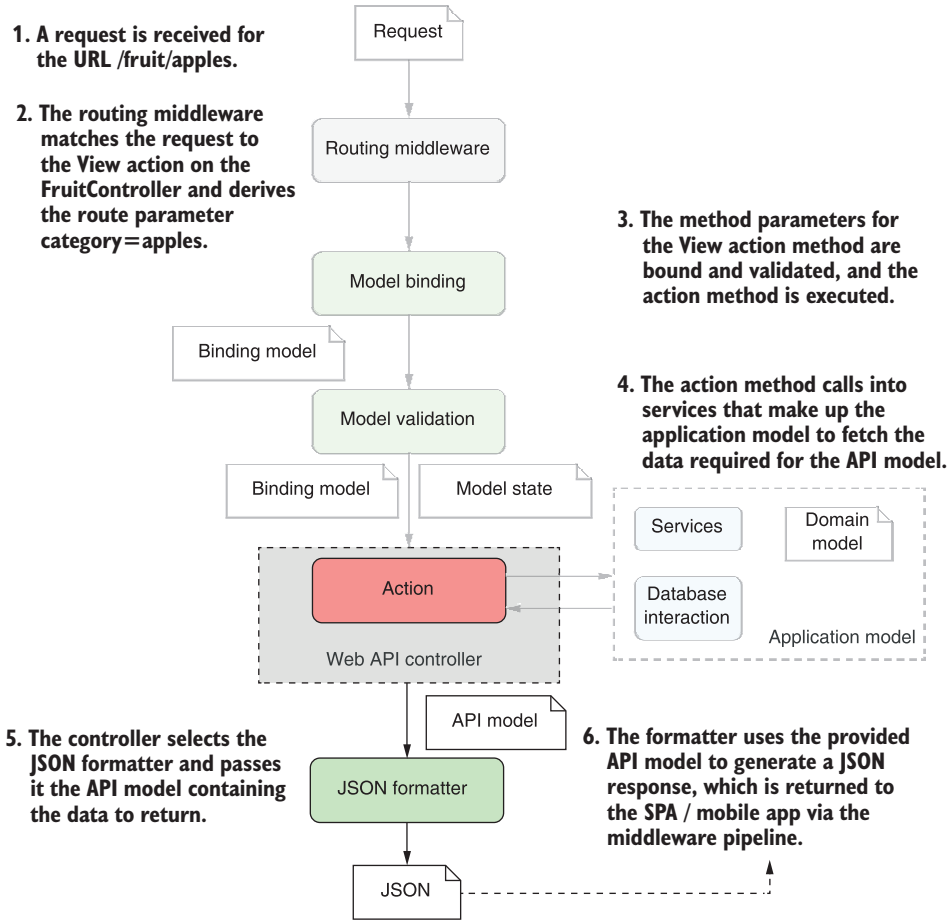

Figure 9.9 A call to a Web API endpoint in an e-commerce ASP.NET Core web application. The ghosted portion of the diagram is identical to figure 9.8.

After the application model has returned the data necessary to service the request—the fruit objects in the `apples` category—you see the first significant difference between API controllers and Razor Pages. Instead of adding values to the `PageModel` to be used in a Razor view, the action method creates an *API model*. This is analogous to the `PageModel`, but rather than containing data used to generate an HTML view, it contains the data that will be sent back in the response.

> **DEFINITION** *View models* and `PageModels` contain both the *data* required to build a response and *metadata* about *how* to build the response. API models typically *only* contain the data to be returned in the response.

When we looked at the Razor Pages app, we used the `PageModel` in conjunction with a Razor view template to build the final response. With the Web API app, we use the API

model in conjunction with an *output formatter*. An output formatter, as the name suggests, serializes the API model into a machine-readable response, such as JSON or XML. The output formatter forms the "V" in the Web API version of MVC, by choosing an appropriate representation of the data to return.

Finally, as for the Razor Pages app, the generated response is then sent back through the middleware pipeline, passing through each of the configured middleware components, and back to the original caller.

Hopefully, the parallels between Razor Pages and Web APIs are clear; the majority of the behavior is identical—only the response varies. Everything from when the request arrives to the interaction with the application model is similar between the paradigms.

Most of the differences between Razor Pages and Web APIs have less to do with the way the framework works under the hood and are instead related to how the different paradigms are used. For example, in the next section, you'll learn how the routing constructs you learned about in chapter 5 are used with Web APIs, using attribute routing.

9.4 Attribute routing: Linking action methods to URLs

In this section you'll learn about attribute routing: the mechanism for associating API controller actions with a given route template. You'll see how to associate controller actions with specific HTTP verbs like `GET` and `POST` and how to avoid duplication in your templates.

We covered route templates in depth in chapter 5 in the context of Razor Pages, and you'll be pleased to know that you use exactly the same route templates with API controllers. The only difference is how you *specify* the templates: with Razor Pages you use the `@page` directive, whereas with API controllers you use routing attributes.

> **NOTE** Both Razor Pages and API controllers use *attribute routing* under the hood. The alternative, *conventional* routing, is typically used with traditional MVC controllers and views. As discussed previously, I don't recommend using that approach, so I don't cover conventional routing in this book.

With attribute routing, you decorate each action method in an API controller with an attribute and provide the associated route template for the action method, as shown in the following listing.

Listing 9.4 Attribute routing example

```
public class HomeController: Controller
{
    [Route("")]
    public IActionResult Index()
    {
        /* method implementation*/
    }
```

The Index action will be executed when the / URL is requested.

```
    [Route("contact")]
    public IActionResult Contact()
    {
        /* method implementation*/
    }
}
```

The Contact action will be executed when the /contact URL is requested.

Each `[Route]` attribute defines a route template that should be associated with the action method. In the example provided, the / URL maps directly to the `Index` method and the `/contact` URL maps to the `Contact` method.

Attribute routing maps URLs to a specific action method, but a single action method can still have multiple route templates and hence can correspond to multiple URLs. Each template must be declared with its own `RouteAttribute`, as shown in this listing, which shows the skeleton of a Web API for a car-racing game.

Listing 9.5 Attribute routing with multiple attributes

```
public class CarController
{
    [Route("car/start")]
    [Route("car/ignition")]
    [Route("start-car")]
    public IActionResult Start()
    {
        /* method implementation*/
    }

    [Route("car/speed/{speed}")]
    [Route("set-speed/{speed}")]
    public IActionResult SetCarSpeed(int speed)
    {
        /* method implementation*/
    }
}
```

The Start method will be executed when any of these route templates are matched.

The name of the action method has no effect on the route template.

The RouteAttribute template can contain route parameters, in this case {speed}.

The listing shows two different action methods, both of which can be accessed from multiple URLs. For example, the `Start` method will be executed when any of the following URLs are requested:

- `/car/start`
- `/car/ignition`
- `/start-car`

These URLs are completely independent of the controller and action method names; only the value in the `RouteAttribute` matters.

NOTE By default, the controller and action name have no bearing on the URLs or route templates when `RouteAttributes` are used.

The templates used in route attributes are standard route templates, the same as you used in chapter 5. You can use literal segments and you're free to define route

parameters that will extract values from the URL, as shown by the `SetCarSpeed` method in the previous listing. That method defines two route templates, both of which define a route parameter, `{speed}`.

> **TIP** I've used multiple `[Route]` attributes on each action in this example, but it's best practice to expose your action at a *single* URL. This will make your API easier to understand and for other applications to consume.

Route parameters are handled in the very same way as for Razor Pages—they represent a segment of the URL that can vary. As for Razor Pages, the route parameters in your `RouteAttribute` templates can

- Be optional
- Have default values
- Use route constraints

For example, you could update the `SetCarSpeed` method in the previous listing to constrain `{speed}` to an integer and to default to `20` like so:

```
[Route("car/speed/{speed=20:int}")]
[Route("set-speed/{speed=20:int}")]
public IActionResult SetCarSpeed(int speed)
```

> **NOTE** As discussed in chapter 5, don't use route constraints for validation. For example, if you call the preceding `"set-speed/{speed=20:int}"` route with an invalid value for speed, `/set-speed/oops`, you will get a `404 Not Found` response, as the route does not match. Without the `int` constraint, you would receive the more sensible `400 Bad Request` response.

If you managed to get your head around routing in chapter 5, then routing with API controllers shouldn't hold any surprises for you. One thing you might begin noticing when you start using attribute routing with API controllers is the amount you repeat yourself. Razor Pages removes a lot of repetition by using conventions to calculate route templates based on the Razor Page's filename.

Luckily there are a few features available to make your life a little easier. In particular, combining route attributes and token replacement can help reduce duplication in your code.

9.4.1 Combining route attributes to keep your route templates DRY

Adding route attributes to all of your API controllers can get a bit tedious, especially if you're mostly following conventions where your routes have a standard prefix, such as `"api"` or the controller name. Generally, you'll want to ensure you don't repeat yourself (DRY) when it comes to these strings. The following listing shows two action methods with a number of `[Route]` attributes. (This is for demonstration purposes only. Stick to one per action if you can!)

Listing 9.6 Duplication in `RouteAttribute` templates

```
public class CarController
{
    [Route("api/car/start")]
    [Route("api/car/ignition")]
    [Route("/start-car")]
    public IActionResult Start()
    {
        /* method implementation*/
    }

    [Route("api/car/speed/{speed}")]
    [Route("/set-speed/{speed}")]
    public IActionResult SetCarSpeed(int speed)
    {
        /* method implementation*/
    }
}
```

Multiple route templates use the same "api/car" prefix.

There's quite a lot of duplication here—you're adding `"api/car"` to most of your routes. Presumably, if you decided to change this to `"api/vehicles"`, you'd have to go through each attribute and update it. Code like that is asking for a typo to creep in!

To alleviate this pain, it's possible to apply `RouteAttributes` to controllers, in addition to action methods, as you saw briefly in chapter 5. When a controller and an action method both have a route attribute, the overall route template for the method is calculated by combining the two templates.

Listing 9.7 Combining `RouteAttribute` templates

```
[Route("api/car")]
public class CarController
{
    [Route("start")]
    [Route("ignition")]
    [Route("/start-car")]
    public IActionResult Start()
    {
        /* method implementation*/
    }

    [Route("speed/{speed}")]
    [Route("/set-speed/{speed}")]
    public IActionResult SetCarSpeed(int speed)
    {
        /* method implementation*/
    }
}
```

Combines to give the "api/car/start" template

Combines to give the "api/car/ignition" template

Does not combine because it starts with /; gives the "start-car" template

Combines to give the "api/car/speed/{speed}" template

Does not combine because it starts with /; gives the "set-speed/{speed}" template

Combining attributes in this way can reduce some of the duplication in your route templates and makes it easier to add or change the prefixes (such as switching `"car"` to `"vehicle"`) for multiple action methods. To ignore the `RouteAttribute` on the

controller and create an absolute route template, start your action method route template with a slash (`/`). Using a controller `RouteAttribute` reduces a lot of the duplication, but you can do one better by using token replacement.

9.4.2 Using token replacement to reduce duplication in attribute routing

The ability to combine attribute routes is handy, but you're still left with some duplication if you're prefixing your routes with the name of the controller, or if your route templates always use the action name. If you wish, you can simplify even further!

Attribute routes support the automatic replacement of the `[action]` and `[controller]` tokens in your attribute routes. These will be replaced with the name of the action and the controller (without the "Controller" suffix), respectively. The tokens are replaced after all attributes have been combined, so this is useful when you have controller inheritance hierarchies. This listing shows how you can create a `BaseController` class that you can use to apply a consistent route template prefix to *all* the API controllers in your application.

Listing 9.8 Token replacement in `RouteAttributes`

```
[Route("api/[controller]")]
public abstract class BaseController { }

public class CarController : BaseController
{
    [Route("[action]")]
    [Route("ignition")]
    [Route("/start-car")]
    public IActionResult Start()
    {
        /* method implementation*/
    }
}
```

You can apply attributes to a base class, and derived classes will inherit them.

Token replacement happens last, so [controller] is replaced with "car" not "base".

Combines and replaces tokens to give the "api/car/start" template

Combines and replaces tokens to give the "api/car/ignition" template

Does not combine with base attributes because it starts with /, so it remains as "start-car"

WARNING If you use token replacement for `[controller]` or `[action]`, remember that renaming classes and methods will change your public API. If that worries you, you can stick to using static strings like `"car"` instead.

When combined with everything you learned in chapter 5, we've covered pretty much everything there is to know about attribute routing. There's just one more thing to consider: handling different HTTP request types like `GET` and `POST`.

9.4.3 Handling HTTP verbs with attribute routing

In Razor Pages, the HTTP verb, such as `GET` or `POST`, isn't part of the routing process. The `RoutingMiddleware` determines which Razor Page to execute based solely on the route template associated with the Razor Page. It's only when a Razor Page is about to

be executed that the HTTP verb is used to decide which page handler to execute: `OnGet` for the `GET` verb, or `OnPost` for the `POST` verb, for example.

With API controllers, things work a bit differently. For API controllers, the HTTP verb takes part in the routing process itself, so a `GET` request may be routed to one action, and a `POST` request may be routed to a different action, *even though the request used the same URL*. This pattern, where the HTTP verb is an important part of routing, is common in HTTP API design.

For example, imagine you're building an API to manage your calendar. You want to be able to list and create appointments. Well, a traditional HTTP REST service might define the following URLs and HTTP verbs to achieve this:

- `GET /appointments`—List all your appointments
- `POST /appointments`—Create a new appointment

Note that these two endpoints use the same URL; only the HTTP verb differs. The `[Route]` attribute we've used so far responds to *all* HTTP verbs, which is no good for us here—we want to select a different action based on the combination of the URL *and* the verb. This pattern is common when building Web APIs and, luckily, it's easy to model in ASP.NET Core.

ASP.NET Core provides a set of attributes that you can use to indicate which verb an action should respond to. For example,

- `[HttpPost]` handles `POST` requests only.
- `[HttpGet]` handles `GET` requests only.
- `[HttpPut]` handles `PUT` requests only.

There are similar attributes for all the standard HTTP verbs, like `DELETE` and `OPTIONS`. You can use these attributes instead of the `[Route]` attribute to specify that an action method should correspond to a single verb, as shown in the following listing.

Listing 9.9 Using HTTP verb attributes with attribute routing

```
public class AppointmentController
{
    [HttpGet("/appointments")]
    public IActionResult ListAppointments()
    {
        /* method implementation */
    }

    [HttpPost("/appointments")]
    public IActionResult CreateAppointment()
    {
        /* method implementation */
    }
}
```

Only executed in response to GET /appointments

Only executed in response to POST /appointments

If your application receives a request that matches the route template of an action method, but which *doesn't* match the required HTTP verb, you'll get a `405 Method not`

allowed error response. For example, if you send a DELETE request to the /appointments URL in the previous listing, you'll get a 405 error response.

Attribute routing has been used with API controllers since the first days of ASP.NET Core, as it allows tight control over the URLs that your application exposes. When you're building API controllers, there is some code that you'll find yourself writing repeatedly. The [ApiController] attribute, introduced in ASP.NET Core 2.1, is designed to handle some of this for you and reduce the amount of boilerplate you need.

9.5 Using common conventions with the [ApiController] attribute

In this section you'll learn about the [ApiController] attribute and how it can reduce the amount of code you need to write to create consistent Web API controllers. You'll learn about the conventions it applies, why they're useful, and how to turn them off if you need to.

The [ApiController] attribute was introduced in .NET Core 2.1 to simplify the process of creating Web API controllers. To understand what it does, it's useful to look at an example of how you might write a Web API controller *without* the [ApiController] attribute, and compare that to the code required to achieve the same thing with the attribute.

Listing 9.10 Creating a Web API controller without the [ApiController] attribute

```
public class FruitController : ControllerBase
{
    List<string> _fruit = new List<string>
    {
        "Pear", "Lemon", "Peach"
    };

    [HttpPost("fruit")]
    public ActionResult Update([FromBody] UpdateModel model)
    {
        if (!ModelState.IsValid)
        {
            return BadRequest(
               new ValidationProblemDetails(ModelState));
        }

        if (model.Id < 0 || model.Id > _fruit.Count)
        {
            return NotFound(new ProblemDetails()
            {
                Status = 404,
                Title = "Not Found",
                Type = "https://tools.ietf.org/html/rfc7231"
                       + "#section-6.5.4",
            });
        }
```

Web APIs use attribute routing to define the route templates.

The [FromBody] attribute indicates that the parameter should be bound to the request body.

The list of strings serves as the application model in this example.

You need to check if model validation succeeded and return a 400 response if it failed.

If the data sent does not contain a valid ID, return a 404 ProblemDetails response.

```
        _fruit[model.Id] = model.Name;
        return Ok();
    }

    public class UpdateModel
    {
        public int Id { get; set; }

        [Required]
        public string Name { get; set; }
    }
}
```

Update the model and return a 200 Response.

UpdateModel is only valid if the Name value is provided, as set by the [Required] attribute.

This example demonstrates many common features and patterns used with Web API controllers:

- Web API controllers read data from the body of a request, typically sent as JSON. To ensure the body is read as JSON and not as form values, you have to apply the `[FromBody]` attribute to the method parameters to ensure it is model-bound correctly.
- As discussed in chapter 6, after model-binding, the model is validated, but it's up to you to act on the validation results. You should return a `400 Bad Request` response if the values provided failed validation. You typically want to provide details of *why* the request was invalid: this is done in listing 9.10 by returning a `ValidationProblemDetails` object in the response body, built from the `ModelState`.
- Whenever you return an error status, such as a `404 Not Found`, where possible you should return details of the problem that will allow the caller to diagnose the issue. The `ProblemDetails` class is the recommended way of doing that in ASP.NET Core.

The code in listing 9.10 is representative of what you might see in an ASP.NET Core API controller *prior* to .NET Core 2.1. The introduction of the `[ApiController]` attribute in .NET Core 2.1 (and subsequent refinement in later versions), makes this same code much simpler, as shown in the following listing.

Listing 9.11 Creating a Web API controller with the `[ApiController]` attribute

```
[ApiController]
public class FruitController : ControllerBase
{
    List<string> _fruit = new List<string>
    {
        "Pear", "Lemon", "Peach"
    };

    [HttpPost("fruit")]
    public ActionResult Update(UpdateModel model)
    {
        if (model.Id < 0 || model.Id > _fruit.Count)
```

Adding the [ApiController] attribute applies several conventions common to API controllers.

The [FromBody] attribute is assumed for complex action method parameters.

The model validation is automatically checked, and if invalid, returns a 400 response.

```
            {
                return NotFound();
            }

            _fruit[model.Id] = model.Name;

            return Ok();
        }

        public class UpdateModel
        {
            public int Id { get; set; }

            [Required]
            public string Name { get; set; }
        }
    }
```

Error status codes are automatically converted to a ProblemDetails object.

If you compare listing 9.10 to listing 9.11, you'll see that all the bold code in listing 9.10 can be removed and replaced with the `[ApiController]` attribute in listing 9.11. The `[ApiController]` attribute automatically applies several conventions to your controllers:

- *Attribute routing*—You must use attribute routing with your controllers; you can't use conventional routing. Not that you would, as we've only discussed this approach for API controllers anyway.
- *Automatic 400 responses*—I said in chapter 6 that you should *always* check the value of `ModelState.IsValid` in your Razor Page handlers and MVC actions, but the `[ApiController]` attribute does this for you by adding a *filter*. We'll cover filters in detail in chapter 13.
- *Model binding source inference*—Without the `[ApiController]` attribute, complex types are assumed to be passed as *form* values in the request body. For Web APIs, it's much more common to pass data as JSON, which ordinarily requires adding the `[FromBody]` attribute. The `[ApiController]` attribute takes care of that for you.
- `ProblemDetails` *for error codes*—You often want to return a consistent set of data when an error occurs in your API. `ProblemDetails` is a type based on a web standard that serves as this consistent data. The `[ApiController]` attribute will intercept any error status codes returned by your controller (for example, a `404 Not Found` response), and convert them into the `ProblemDetails` type automatically.

A key feature of the `[ApiController]` attribute is using the Problem Details format to return errors in a consistent format across all your controllers.[3] A typical `ProblemDetails` object looks something like the following, which shows an example `ValidationProblemDetails` object generated automatically for an invalid request:

[3] Problem Details is a proposed standard for handling errors in a machine-readable way that is gaining popularity. You can find the specification here, but be warned, it makes for dry reading: https://tools.ietf.org/html/rfc7807.

```
{
  type: "https://tools.ietf.org/html/rfc7231#section-6.5.1"
  title: "One or more validation errors occurred."
  status: 400
  traceId: "|17a2706d-43736ad54bed2e65."
  errors: {
    name: ["The name field is required."]
  }
}
```

The `[ApiController]` conventions can significantly reduce the amount of boilerplate code you have to write. They also ensure consistency across your whole application. For example, you can be sure that all your controllers will return the same error type, `ValidationProblemDetails` (a sub-type of `ProblemDetails`), when a bad request is received.

Converting all your errors to `ProblemDetails`

The `[ApiController]` attribute ensures that all error responses returned by your API controllers are converted into `ProblemDetails` objects, which keeps the error responses consistent across your application.

The only problem with this is that your API controllers aren't the *only* thing that could generate errors. For example, if a URL is received that doesn't match any action in your controllers, the end-of-the-pipeline middleware we discussed in chapter 3 would generate a `404 Not Found` response. As this error is generated *outside* of the API controllers, it won't use `ProblemDetails`. Similarly, when your code throws an exception, you want this to be returned as a `ProblemDetails` object too, but this doesn't happen by default.

In chapter 3 I described several types of error handling middleware that you could use to handle these cases, but it can be complicated to handle all the edge cases. I prefer to use a community-created package, Hellang.Middleware.ProblemDetails, which takes care of this for you. You can read about how to use this package on my blog at http://mng.bz/Gx7D.

As is common in ASP.NET Core, you will be most productive if you follow the conventions rather than trying to fight them. However, if you don't like some of the conventions, or want to customize them, you can easily do so.

You can customize the conventions your application uses by calling `ConfigureApiBehaviorOptions()` on the `IMvcBuilder` object returned from the `AddControllers()` method in your Startup.cs file. For example, you could disable the automatic `400` responses on validation failure, as shown in the following listing.[4]

[4] You can disable all the automatic features enabled by the `[ApiController]` attribute, but I'd encourage you to stick to the defaults unless you really need to change them. You can read more about disabling features in the documentation at https://docs.microsoft.com/aspnet/core/web-api.

Listing 9.12 Customizing `[ApiAttribute]` behaviors

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddControllers()
            .ConfigureApiBehaviorOptions(options =>
            {
                options.SuppressModelStateInvalidFilter = true;
            })
    }

    // ...
}
```

Control which conventions are applied by providing a configuration lambda.

This would disable the automatic 400 responses for invalid requests.

The ability to customize each aspect of ASP.NET Core is one of the features that sets it apart from the previous version of ASP.NET. ASP.NET Core configures the vast majority of its internal components using one of two mechanisms—dependency injection or by configuring an `Options` object when you add the service to your application, as you'll see in chapters 10 (dependency injection) and 11 (`Options` object).

In the next section you'll learn how to control the format of the data returned by your Web API controllers—whether that's JSON, XML, or a different, custom format.

9.6 Generating a response from a model

This brings us to the final topic in this chapter: formatting a response. It's common for API controllers to return JSON these days, but that's not *always* the case. In this section you'll learn about content negotiation and how to enable additional output formats such as XML. You'll also learn about an important change in the JSON formatter for ASP.NET Core 3.0.

Consider this scenario: You've created a Web API action method for returning a list of cars, as in the following listing. It invokes a method on your application model, which hands back the list of data to the controller. Now you need to format the response and return it to the caller.

Listing 9.13 A Web API controller to return a list of cars

```
[ApiController]
public class CarsController : Controller
{
    [HttpGet("api/cars")]
    public IEnumerable<string> ListCars()
    {
        return new string[]
            { "Nissan Micra", "Ford Focus" };
    }
}
```

The action is executed with a request to GET /api/cars.

The API model containing the data is an IEnumerable<string>.

This data would normally be fetched from the application model.

You saw in section 9.2 that it's possible to return data directly from an action method, in which case the middleware formats it and returns the formatted data to the caller. But how does the middleware know which format to use? After all, you could serialize it as JSON, as XML, or even with a simple `ToString()` call.

The process of determining the format of data to send to clients is known generally as *content negotiation* (conneg). At a high level, the client sends a header indicating the types of content it can understand—the `Accept` header—and the server picks one of these, formats the response, and sends a `content-type` header in the response, indicating which type it chose.

The `Accept` and `content-type` headers

The `Accept` header is sent by a client as part of a request to indicate the type of content that the client can handle. It consists of a number of MIME types,[a] with optional weightings (from 0 to 1) to indicate which type would be preferred. For example, the `application/json,text/xml;q=0.9,text/plain;q=0.6` header indicates that the client can accept JSON, XML, and plain text, with weightings of 1.0, 0.9, and 0.6, respectively. JSON has a weighting of 1.0, as no explicit weighting was provided. The weightings can be used during content negotiation to choose an optimal representation for both parties.

The `content-type` header describes the data sent in a request or response. It contains the MIME type of the data, with an optional character encoding. For example, the `application/json; charset=utf-8` header would indicate that the body of the request or response is JSON, encoded using UTF-8.

[a] For more on MIME types, see the Mozilla documentation: http://mng.bz/gop8.

You're not forced into *only* sending a `content-type` the client expects, and, in some cases, you may not even be *able* to handle the types it requests. What if a request stipulates it can only accept Excel spreadsheets? It's unlikely you'd support that, even if that's the only `content-type` the request contains.

When you return an API model from an action method, whether directly (as in listing 9.13) or via an `OkResult` or other `StatusCodeResult`, ASP.NET Core will always return *something*. If it can't honor any of the types stipulated in the `Accept` header, it will fall back to returning JSON by default. Figure 9.10 shows that even though XML was requested, the API controller formatted the response as JSON.

NOTE In the previous version of ASP.NET, objects were serialized to JSON using PascalCase, where properties start with a capital letter. In ASP.NET Core, objects are serialized using camelCase by default, where properties start with a lowercase letter.

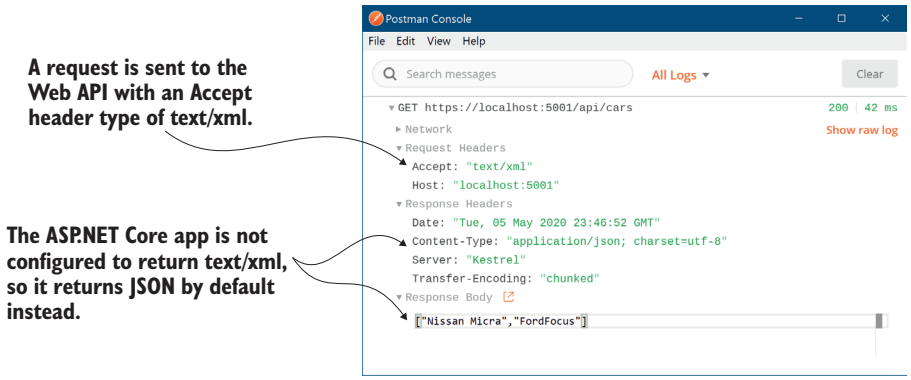

Figure 9.10 Even though the request was made with an `Accept` header of `text/xml`, the response returned was JSON, as the server was not configured to return XML.

However the data is sent, it's serialized by an `IOutputFormatter` implementation. ASP.NET Core ships with a limited number of output formatters out of the box, but as always, it's easy to add additional ones or change the way the defaults work.

9.6.1 Customizing the default formatters: Adding XML support

As with most of ASP.NET Core, the Web API formatters are completely customizable. By default, only formatters for plain text (`text/plain`), HTML (`text/html`), and JSON (`application/json`) are configured. Given the common use case of SPAs and mobile applications, this will get you a long way. But sometimes you need to be able to return data in a different format, such as XML.

Newtonsoft.Json vs. System.Text.Json

Newtonsoft.Json, also known as Json.NET, has for a long time been the canonical way to work with JSON in .NET. It's compatible with every version of .NET under the sun, and it will no doubt be familiar to virtually all .NET developers. Its reach was so great that even ASP.NET Core took a dependency on it!

That all changed with the introduction of a new library in ASP.NET Core 3.0, System.Text.Json, which focuses on performance. In ASP.NET Core 3.0 onwards, ASP.NET Core uses System.Text.Json by default instead of Newtonsoft.Json.

The main difference between the libraries is that System.Text.Json is very picky about its JSON. It will generally only deserialize JSON that matches its expectations. For example, System.Text.Json won't deserialize JSON that uses single quotes around strings; you have to use double quotes.

If you're creating a new application, this is generally not a problem—you quickly learn to generate the correct JSON. But if you're migrating an application from ASP.NET Core 2.0 or are receiving JSON from a third party, these limitations can be real stumbling blocks.

Luckily, you can easily switch back to the Newtonsoft.Json library instead. Install the Microsoft.AspNetCore.Mvc.NewtonsoftJson package into your project and update the `AddControllers()` method in Startup.cs to the following:

```
services.AddControllers()
    .AddNewtonsoftJson();
```

This will switch ASP.NET Core's formatters to use Newtonsoft.Json behind the scenes, instead of System.Text.Json. For more details on the differences between the libraries, see Microsoft's article "How to migrate from Newtonsoft.Json to System.Text.Json": http://mng.bz/0mRJ. For more advice on when to switch to the Newtonsoft.Json formatter, see the section "Add Newtonsoft.Json-based JSON format support" in Microsoft's "Format response data in ASP.NET Core Web API" documentation: http://mng.bz/zx11.

You can add XML output to your application by adding an *output formatter*. You configure your application's formatters in Startup.cs, by customizing the `IMvcBuilder` object returned from `AddControllers()`. To add the XML output formatter,[5] use the following:

```
services.AddControllers()
    .AddXmlSerializerFormatters();
```

With this simple change, your API controllers can now format responses as XML. Running the same request as shown in figure 9.10 with XML support enabled means the app will respect the `text/xml` `accept` header. The formatter serializes the `string` array to XML as requested instead of defaulting to JSON, as shown in figure 9.11.

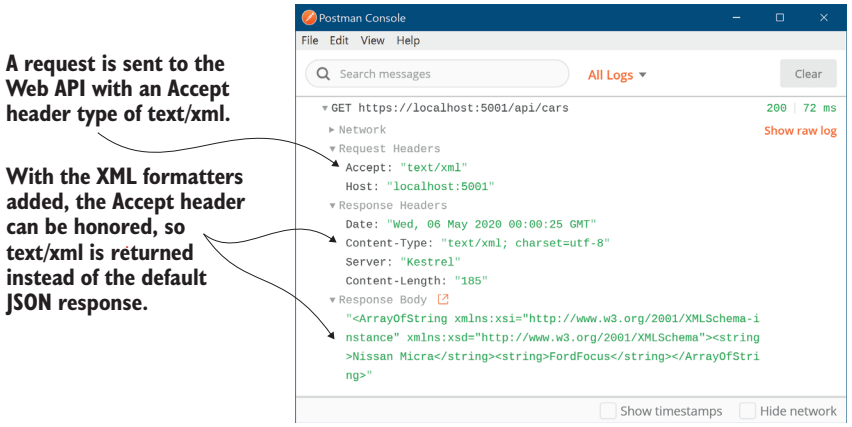

Figure 9.11 With the XML output formatters added, the `text/xml` `Accept` header is respected and the response can be serialized to XML.

[5] Technically this also adds an XML *input formatter* as well, which means your application can now *receive* XML in requests too. For a detailed look at formatters, including creating a custom formatter, see the documentation at http://mng.bz/e5gG.

This is an example of content negotiation, where the client has specified what formats it can handle and the server selects one of those, based on what it can produce. This approach is part of the HTTP protocol, but there are some quirks to be aware of when relying on it in ASP.NET Core. You won't often run into these, but if you're not aware of them when they hit you, they could have you scratching your head for hours!

9.6.2 Choosing a response format with content negotiation

Content negotiation is where a client says which types of data it can accept using the `Accept` header and the server picks the best one it can handle. Generally speaking, this works as you'd hope: the server formats the data using a type the client can understand.

The ASP.NET Core implementation has some special cases that are worth bearing in mind:

- By default, ASP.NET Core will only return `application/json`, `text/plain`, and `text/html` MIME types. You can add additional `IOutputFormatters` to make other types available, as you saw in the previous section for `text/xml`.
- By default, if you return `null` as your API model, whether from an action method or by passing `null` in a `StatusCodeResult`, the middleware will return a `204 No Content` response.
- When you return a `string` as your API model, if no `Accept` header is set, the middleware will format the response as `text/plain`.
- When you use any other class as your API model and there's either no `Accept` header or none of the supported formats were requested, the first formatter that can generate a response will be used (typically JSON by default).
- If the middleware detects that the request is probably from a browser (the `accept` header contains `*/*`), then it will *not* use conneg. Instead, it will format the response as though no `Accept` header was provided, using the default formatter (typically JSON).

These defaults are relatively sane, but they can certainly bite you if you're not aware of them. The last point in particular, where the response to a request from a browser is virtually always formatted as JSON, has certainly caught me out when trying to test XML requests locally!

As you should expect by now, all these rules are configurable; you can easily change the default behavior in your application if it doesn't fit your requirements. For example, the following listing, taken from Startup.cs, shows how you can force the middleware to respect the browser's `Accept` header, and remove the `text/plain` formatter for `strings`.

Listing 9.14 Customizing MVC to respect the browser's `Accept` header in Web APIs

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers(options =>
```

AddControllers has an overload that takes a lambda function.

```
    {
        options.RespectBrowserAcceptHeader = true;
        options.OutputFormatters.RemoveType<StringOutputFormatter>();
    });
}
```

False by default, a number of other properties are also available to be set.

Removes the output formatter that formats strings as text/plain

In most cases, conneg should work well for you out of the box, whether you're building an SPA or a mobile application. In some cases, you may find you need to bypass the usual conneg mechanisms for specific action methods, and there are a number of ways to achieve this, but I won't cover them in this book as I've found I rarely need to use them. For details, see Microsoft's "Format response data in ASP.NET Core Web API" documentation: http://mng.bz/zx11.

That brings us to the end of this chapter on Web APIs and, with it, part 1 of this book! It's been a pretty intense tour of ASP.NET Core, with a heavy focus on Razor Pages and the MVC pattern. By making it this far, you now have all the knowledge you need to start building simple applications using Razor Pages or to create a Web API server for your SPA or mobile app.

In part 2 you'll get into some juicy topics: you'll learn the details needed to build complete apps, like adding users to your application, saving data to a database, and deploying your application.

In chapter 10 we'll look at dependency injection in ASP.NET Core and how it helps create loosely coupled applications. You'll learn how to register the ASP.NET Core framework services with a container and set up your own classes as dependency-injected services. Finally, you'll see how to replace the built-in container with a third-party alternative.

Summary

- A Web API exposes a number of methods or endpoints that can be used to access or change data on a server. It's typically accessed using HTTP by mobile or client-side web applications.
- Web API action methods can return data directly or can use `ActionResult<T>` to generate an arbitrary response.
- If you return more than one type of result from an action method, the method signature must return `ActionResult<T>`.
- Web APIs follow the same MVC design pattern as traditional web applications. The formatters that generate the final response form the view.
- The data returned by a Web API action is called an API model. It contains the data the middleware will serialize and send back to the client. This differs from view models and `PageModels`, which contain both data and metadata about how to generate the response.

- Web APIs are associated with route templates by applying `RouteAttributes` to your action methods. These give you complete control over the URLs that make up your application's API.
- Route attributes applied to a controller combine with attributes on action methods to form the final template. These are also combined with attributes on inherited base classes. You can use inherited attributes to reduce the amount of duplication in the attributes, such as where you're using a common prefix on your routes.
- By default, the controller and action name have no bearing on the URLs or route templates when you use attribute routing. However, you can use the `"[controller]"` and `"[action]"` tokens in your route templates to reduce repetition. They'll be replaced with the current controller and action name.
- The `[HttpPost]` and `[HttpGet]` attributes allow you to choose between actions based on the request's HTTP verb when two actions correspond to the same URL. This is a common pattern in RESTful applications.
- The `[ApiController]` attribute applies several common conventions to your controllers. Controllers decorated with the attribute will automatically bind to a request's body instead of using form values, will automatically generate a `400 Bad Request` response for invalid requests, and will return `ProblemDetails` objects for status code errors. This can dramatically reduce the amount of boilerplate code you must write.
- You can control which of the conventions to apply by using the `ConfigureApiBehaviorOptions()` method and providing a configuration lambda. This is useful if you need to fit your API to an existing specification, for example.
- By default, ASP.NET Core formats the API model returned from a Web API controller as JSON. Virtually every platform can handle JSON, making your API highly interoperable.
- In contrast to the previous version of ASP.NET, JSON data is serialized using camelCase rather than PascalCase. You should consider this change if you get errors or missing values when migrating from ASP.NET to ASP.NET Core.
- ASP.NET Core 3.0 onwards uses System.Text.Json, which is a strict, high performance library for JSON serialization and deserialization. You can replace this serializer with the common Newtonsoft.Json formatter by calling `AddNewtonsoftJson()` on the return value from `services.AddControllers()`.
- Content negotiation occurs when the client specifies the type of data it can handle and the server chooses a return format based on this. It allows multiple clients to call your API and receive data in a format they can understand.
- By default, ASP.NET Core can return `text/plain`, `text/html`, and `application/json`, but you can add additional formatters if you need to support other formats.

- You can add XML formatters by calling `AddXmlSerializerFormatters()` on the return value from `services.AddControllers()` in your `Startup` class. These let you format the response as XML, as well as receive XML in a request body.
- Content negotiation isn't used when the `Accept` header contains `*/*`, such as in most browsers. Instead, your application will use the default formatter, JSON. You can disable this option by modifying the `RespectBrowserAcceptHeader` option when adding your MVC controller services in Startup.cs.